**Quality of Services**

Each policy defines a structure to specify its data. Each entity supports a subset of the policies and defines a QoS structure that is composed of the supported policy structures.

For example, the Publisher’s QoS structure is defined in the specification’s IDL as follows:

module DDS

{

struct PublisherQos

{

PresentationQosPolicy presentation;

PartitionQosPolicy partition;

GroupDataQosPolicy group\_data;

EntityFactoryQosPolicy entity\_factory;

};

};

Setting policies is as simple as obtaining a structure with the default values already set, modifying the individual policy structures as necessary, and then applying the QoS structure to an entity.

Applications can change the QoS of any entity by calling the set\_qos() operation on the entity. If the QoS is changeable, existing associations are removed if they are no longer compatible and new associations are added if they become compatible.

A subset of the QoS policies are changeable. Some changeable QoS policies, such as USER\_DATA, TOPIC\_DATA, GROUP\_DATA, LIFESPAN, OWNERSHIP\_STRENGTH, TIME\_BASED\_FILTER, ENTITY\_FACTORY, WRITER\_DATA\_LIFECYCLE, and READER\_DATA\_LIFECYCLE, do not require compatibility and association re-evaluation. The DEADLINE and LATENCY\_BUDGET QoS policies require compatibility re-evaluation, but not for association. The PARTITION QoS policy does not require compatibility re-evaluation but does require association re-evaluation. The DDS specification lists TRANSPORT\_PRIORITY as changeable, but the OpenDDS implementation does not support dynamically modifying this policy.

**Default QoS**

Applications obtain the default QoS policies for an entity by instantiating a QoS structure of the appropriate type for the entity and passing it by reference to the appropriate get\_default\_entity\_qos() operation on the appropriate factory entity.

The following examples illustrate how to obtain the default policies for publisher, subscriber, topic, domain participant, data writer, and data reader.

* // Get default Publisher QoS from a DomainParticipant: DDS::PublisherQos pub\_qos; DDS::ReturnCode\_t ret; ret = domain\_participant->get\_default\_publisher\_qos(pub\_qos); if (DDS::RETCODE\_OK != ret) { std::cerr << "Could not get default publisher QoS" << std::endl; }
* // Get default Subscriber QoS from a DomainParticipant: DDS::SubscriberQos sub\_qos; ret = domain\_participant->get\_default\_subscriber\_qos(sub\_qos);

**QoS Policies**

**Liveliness:**  The LIVELINESS policy controls when and how the service determines whether participants are alive, meaning they are still reachable and active. The kind member setting indicates whether liveliness is asserted automatically by the service or manually by the specified entity.

The LIVELINESS policy applies to the topic, data reader, and data writer entities via the liveliness member of their respective QoS structures. Setting this policy on a topic means it is in effect for all data readers and data writers on that topic.

Below is the IDL related to the liveliness QoS policy:

enum LivelinessQosPolicyKind {

AUTOMATIC\_LIVELINESS\_QOS, MANUAL\_BY\_PARTICIPANT\_LIVELINESS\_QOS, MANUAL\_BY\_TOPIC\_LIVELINESS\_QOS

};

struct LivelinessQosPolicy {

LivelinessQosPolicyKind kind;

Duration\_t lease\_duration;

};

The kind member setting indicates whether liveliness is asserted automatically by the service or manually by the specified entity.

A setting of AUTOMATIC\_LIVELINESS\_QOS means that the service will send a liveliness indication if the participant has not sent any network traffic for the lease\_duration. The MANUAL\_BY\_PARTICIPANT\_LIVELINESS\_QOS or MANUAL\_BY\_TOPIC\_LIVELINESS\_QOS setting means the specified entity (data writer for the “by topic” setting or domain participant for the “by participant” setting) must either write a sample or manually assert its liveliness within a specified heartbeat interval. The desired heartbeat interval is specified by the lease\_duration member. The default lease duration is a pre-defined infinite value, which disables any liveliness testing.

To manually assert liveliness without publishing a sample, the application must call the assert\_liveliness() operation on the data writer (for the “by topic” setting) or on the domain participant (for the “by participant” setting) within the specified heartbeat interval.

Compatibility is determined by comparing the data reader’s requested liveliness with the data writer’s offered liveliness. Both the kind of liveliness (automatic, manual by topic, manual by participant) and the value of the lease duration are considered in determining compatibility. The writer’s offered kind of liveliness must be greater than or equal to the reader’s requested kind of liveliness.

In addition, the writer’s offered lease duration must be less than or equal to the reader’s requested lease duration. Both conditions must be met for the offered and requested liveliness policy settings to be considered compatible and the association established.

**Reliability:** This policy controls how data readers and writers treat the data samples they process. The RELIABILITY policy applies to the topic, data reader, and data writer entities via the reliability member of their respective QoS structures. Below is the IDL related to the reliability QoS policy:

enum ReliabilityQosPolicyKind {

BEST\_EFFORT\_RELIABILITY\_QOS,

RELIABLE\_RELIABILITY\_QOS

};

struct ReliabilityQosPolicy {

ReliabilityQosPolicyKind kind;

Duration\_t max\_blocking\_time;

};

The “best effort” value (BEST\_EFFORT\_RELIABILITY\_QOS) makes no promises as to the reliability of the samples and could be expected to drop samples under some circumstances. The “reliable” value (RELIABLE\_RELIABILITY\_QOS) indicates that the service should eventually deliver all values to eligible data readers.

The max\_blocking\_time member of this policy is used when the history QoS policy is set to “keep all” and the writer is unable to proceed because of resource limits. When this situation occurs and the writer blocks for more than the specified time, then the write fails with a timeout return code. The default for this policy for data readers and topics is “best effort,” while the default value for data writers is “reliable.”

This policy is considered during the creation of associations between data writers and data readers. The value of both sides of the association must be compatible for an association to be created. The reliability kind of data writer must be greater than or equal to the value of data reader.

**History:** The HISTORY policy determines how samples are held in the data writer and data reader for a particular instance. For data writers these values are held until the publisher retrieves them and successfully sends them to all connected subscribers. For data readers these values are held until “taken” by the application. This policy applies to the topic, data reader, and data writer entities via the history member of their respective QoS structures. Below is the IDL related to the history QoS policy:

enum HistoryQosPolicyKind {

KEEP\_LAST\_HISTORY\_QOS,

KEEP\_ALL\_HISTORY\_QOS

};

struct HistoryQosPolicy {

HistoryQosPolicyKind kind;

long depth;

};

The “keep all” value (KEEP\_ALL\_HISTORY\_QOS) specifies that all possible samples for that instance should be kept. When “keep all” is specified and the number of unread samples is equal to the “resource limits” field of max\_samples\_per\_instance then any incoming samples are rejected. The “keep last” value (KEEP\_LAST\_HISTORY\_QOS) specifies that only the last depth values should be kept. When a data writer contains depth samples of a given instance, a write of new samples for that instance are queued for delivery and the oldest unsent samples are discarded. When a data reader contains depth samples of a given instance, any incoming samples for that instance are kept and the oldest samples are discarded. This policy defaults to a “keep last” with a depth of one.

**Durability:** The DURABILITY policy controls whether data writers should maintain samples after they have been sent to known subscribers. This policy applies to the topic, data reader, and data writer entities via the durability member of their respective QoS structures. Below is the IDL related to the durability QoS policy:

enum DurabilityQosPolicyKind {

VOLATILE\_DURABILITY\_QOS, // Least Durability

TRANSIENT\_LOCAL\_DURABILITY\_QOS,

TRANSIENT\_DURABILITY\_QOS,

PERSISTENT\_DURABILITY\_QOS // Greatest Durability

};

struct DurabilityQosPolicy { DurabilityQosPolicyKind kind; };

By default the kind is VOLATILE\_DURABILITY\_QOS. A durability kind of VOLATILE\_DURABILITY\_QOS means samples are discarded after being sent to all known subscribers. As a side effect, subscribers cannot recover samples sent before they connect. A durability kind of TRANSIENT\_LOCAL\_DURABILITY\_QOS means that data readers that are associated/connected with a data writer will be sent all of the samples in the data writer’s history. A durability kind of TRANSIENT\_DURABILITY\_QOS means that samples outlive a data writer and last as long as the process is alive. The samples are kept in memory, but are not persisted to permanent storage. A data reader subscribed to the same topic and partition within the same domain will be sent all of the cached samples that belong to the same topic/partition. A durability kind of PERSISTENT\_DURABILITY\_QOS provides basically the same functionality as transient durability except the cached samples are persisted and will survive process destruction.

When transient or persistent durability is specified, the DURABILITY\_SERVICE QoS policy specifies additional tuning parameters for the durability cache.

The durability policy is considered during the creation of associations between data writers and data readers. The value of both sides of the association must be compatible in order for an association to be created. The durability kind value of the data writer must be greater than or equal to the corresponding value of the data reader.

**Durability\_Service:** The DURABILITY\_SERVICE policy controls deletion of samples in TRANSIENT or PERSISTENT durability cache. This policy applies to the topic and data writer entities via the durability\_service member of their respective QoS structures and provides a way to specify HISTORY and RESOURCE\_LIMITS for the sample cache. Below is the IDL related to the durability service QoS policy:

struct DurabilityServiceQosPolicy {

Duration\_t service\_cleanup\_delay;

HistoryQosPolicyKind history\_kind;

long history\_depth;

long max\_samples;

long max\_instances;

long max\_samples\_per\_instance;

};

The service\_cleanup\_delay can be set to a desired value. By default, it is set to zero, which means never clean up cached samples.

**Resource\_Limit:** The RESOURCE\_LIMITS policy determines the amount of resources the service can consume in order to meet the requested QoS. This policy applies to the topic, data reader, and data writer entities via the resource\_limits member of their respective QoS structures. Below is the IDL related to the resource limits QoS policy.

struct ResourceLimitsQosPolicy {

long max\_samples;

long max\_instances;

long max\_samples\_per\_instance;

};

The max\_samples member specifies the maximum number of samples a single data writer or data reader can manage across all of its instances. The max\_instances member specifies the maximum number of instances that a data writer or data reader can manage. The max\_samples\_per\_instance member specifies the maximum number of samples that can be managed for an individual instance in a single data writer or data reader. The values of all these members default to unlimited (DDS::LENGTH\_UNLIMITED). Resources are used by the data writer to queue samples written to the data writer but not yet sent to all data readers because of backpressure from the transport. Resources are used by the data reader to queue samples that have been received, but not yet read/taken from the data reader.

**Partition:** The PARTITION QoS policy allows the creation of logical partitions within a domain. It only allows data readers and data writers to be associated if they have matched partition strings. This policy applies to the publisher and subscriber entities via the partition member of their respective QoS structures. Below is the IDL related to the partition QoS policy:

struct PartitionQosPolicy { StringSeq name; };

The name member defaults to an empty sequence of strings. The default partition name is an empty string and causes the entity to participate in the default partition. The establishment of data reader and data writer associations depends on matching partition strings on the publication and subscription ends. Failure to match partitions is not considered a failure and does not trigger any callbacks or set any status values. The value of this policy may be changed at any time. Changes to this policy may cause associations to be removed or added.

**Deadline:** The DEADLINE QoS policy allows the application to detect when data is not written or read within a specified amount of time. This policy applies to the topic, data writer, and data reader entities via the deadline member of their respective QoS structures. Below is the IDL related to the deadline QoS policy.

struct DeadlineQosPolicy { Duration\_t period; };

The default value of the period member is infinite, which requires no behavior. When this policy is set to a finite value, then the data writer monitors the changes to data made by the application and indicates failure to honor the policy by setting the corresponding status condition and triggering the on\_offered\_deadline\_missed() listener callback. A data reader that detects that the data has not changed before the period has expired sets the corresponding status condition and triggers the on\_requested\_deadline\_missed() listener callback.

This policy is considered during the creation of associations between data writers and data readers. The value of both sides of the association must be compatible in order for an association to be created. The deadline period of the data reader must be greater than or equal to the corresponding value of data writer.

**Lifespan:** The LIFESPAN QoS policy allows the application to specify when a sample expires. Expired samples will not be delivered to subscribers. This policy applies to the topic and data writer entities via the lifespan member of their respective QoS structures. Below is the IDL related to the lifespan QoS policy.

struct LifespanQosPolicy { Duration\_t duration; }

The default value of the duration member is infinite, which means samples never expire. OpenDDS currently supports expired sample detection on the publisher side when using a DURABILITY kind other than VOLATILE. The current OpenDDS implementation may not remove samples from the data writer and data reader caches when they expire after being placed in the cache.

The value of this policy may be changed at any time. Changes to this policy affect only data written after the change.

**User\_data:** The USER\_DATA policy applies to the domain participant, data reader, and data writer entities via the user\_data member of their respective QoS structures. Below is the IDL related to the user data QoS policy:

struct UserDataQosPolicy { sequence value; };

By default, the value member is not set. It can be set to any sequence of octets which can be used to attach information to the created entity. The value of the USER\_DATA policy is available in respective built-in topic data. The remote application can obtain the information via the built-in topic and use it for its own purposes. For example, the application could attach security credentials via the USER\_DATA policy that can be used by the remote application to authenticate the source.

**Topic\_Data:** The TOPIC\_DATA policy applies to topic entities via the topic\_data member of TopicQoS structures. Below is the IDL related to the topic data QoS policy:

struct TopicDataQosPolicy { sequence value; };

By default, the value is not set. It can be set to attach additional information to the created topic. The value of the TOPIC\_DATA policy is available in data writer, data reader, and topic built-in topic data. The remote application can obtain the information via the built-in topic and use it in an application-defined way.

**Transport\_Priority:** The TRANSPORT\_PRIORITY policy applies to topic and data writer entities via the transport\_priority member of their respective QoS policy structures. Below is the IDL related to the TransportPriority QoS policy:

struct TransportPriorityQosPolicy { long value; };

The default value member of transport\_priority is zero. This policy is considered a hint to the transport layer to indicate at what priority to send messages. Higher values indicate higher priority. OpenDDS maps the priority value directly onto thread and DiffServ codepoint values. A default priority of zero will not modify either threads or codepoints in messages.

OpenDDS will attempt to set the thread priority of the sending transport as well as any associated receiving transport.

On most systems, thread priorities can only be set when the process scheduler has been set to allow these operations. Setting the process scheduler is generally a privileged operation and will require system privileges to perform. On POSIX based systems, the system calls of sched\_get\_priority\_min() and sched\_get\_priority\_max() are used to determine the system range of thread priorities.

OpenDDS will attempt to set the DiffServ codepoint on the socket used to send data for the data writer if it is supported by the transport implementation. If the network hardware honors the codepoint values, higher codepoint values will result in better (faster) transport for higher priority samples. The default value of zero will be mapped to the (default) codepoint of zero. Priority values from 1 through 63 are then mapped to the corresponding codepoint values, and higher priority values are mapped to the highest codepoint value (63). OpenDDS does not currently support modifications of the transport\_priority policy values after creation of the data writer. This can be worked around by creating new data writers as different priority values are required.

**Latency\_Budget:** The LATENCY\_BUDGET policy applies to topic, data reader, and data writer entities via the latency\_budget member of their respective QoS policy structures. Below is the IDL related to the LatencyBudget QoS policy:

struct LatencyBudgetQosPolicy { Duration\_t duration; };

The default value of duration is zero indicating that the delay should be minimized. This policy is considered a hint to the transport layer to indicate the urgency of samples being sent.

**Entity\_Factory:** The ENTITY\_FACTORY policy controls whether entities are automatically enabled when they are created. Below is the IDL related to the Entity Factory QoS policy:

struct EntityFactoryQosPolicy { boolean autoenable\_created\_entities; };

This policy can be applied to entities that serve as factories for other entities and controls whether or not entities created by those factories are automatically enabled upon creation. This policy can be applied to the domain participant factory (as a factory for domain participants), domain participant (as a factory for publishers, subscribers, and topics), publisher (as a factory for data writers), or subscriber (as a factory for data readers).

The default value for the autoenable\_created\_entities member is true, indicating that entities are automatically enabled when they are created. Applications that wish to explicitly enable entities some time after they are created should set the value of the autoenable\_created\_entities member of this policy to false and apply the policy to the appropriate factory entities. The application must then manually enable the entity by calling the entity’s enable() operation. The value of this policy may be changed at any time. Changes to this policy affect only entities created after the change.

**Presentation:** The PRESENTATION QoS policy controls how changes to instances by publishers are presented to data readers. It affects the relative ordering of these changes and the scope of this ordering. Additionally, this policy introduces the concept of coherent change sets. Here is the IDL for the Presentation QoS:

enum PresentationQosPolicyAccessScopeKind { INSTANCE\_PRESENTATION\_QOS,

TOPIC\_PRESENTATION\_QOS,

GROUP\_PRESENTATION\_QOS

};

struct PresentationQosPolicy {

PresentationQosPolicyAccessScopeKind access\_scope;

boolean coherent\_access;

boolean ordered\_access; };

The scope of these changes (access\_scope) specifies the level in which an application may be made aware:

– INSTANCE\_PRESENTATION\_QOS (the default) indicates that changes occur to instances independently. Instance access essentially acts as a no-op with respect to coherent\_access and ordered\_access. Setting either of these values to true has no observable affect within the subscribing application.

– TOPIC\_PRESENTATION\_QOS indicates that accepted changes are limited to all instances within the same data reader or data writer.

– GROUP\_PRESENTATION\_QOS indicates that accepted changes are limited to all instances within the same publisher or subscriber.

**Note:** This policy controls the ordering and scope of samples made available to the subscriber, but the subscriber application must use the proper logic in reading samples to guarantee the requested behavior.

**Destination\_Order:** The DESTINATION\_ORDER QoS policy controls the order in which samples within a given instance are made available to a data reader. If a history depth of one (the default) is specified, the instance will reflect the most recent value written by all data writers to that instance. Here is the IDL for the Destination Order Qos:

enum DestinationOrderQosPolicyKind { BY\_RECEPTION\_TIMESTAMP\_DESTINATIONORDER\_QOS, BY\_SOURCE\_TIMESTAMP\_DESTINATIONORDER\_QOS

};

struct DestinationOrderQosPolicy { DestinationOrderQosPolicyKind kind; };

The BY\_RECEPTION\_TIMESTAMP\_DESTINATIONORDER\_QOS value (the default) indicates that samples within an instance are ordered in the order in which they were received by the data reader. Note that samples are not necessarily received in the order sent by the same data writer. To enforce this type of ordering, the BY\_SOURCE\_TIMESTAMP\_DESTINATIONORDER\_QOS value should be used.

The BY\_SOURCE\_TIMESTAMP\_DESTINATIONORDER\_QOS value indicates that samples within an instance are ordered based on a timestamp provided by the data writer. It should be noted that if multiple data writers write to the same instance, care should be taken to ensure that clocks are synchronized to prevent incorrect ordering on the data reader.

**Writer\_Data\_Lifecycle:** The WRITER\_DATA\_LIFECYCLE QoS policy controls the lifecycle of data instances managed by a data writer. Here is the IDL for the Writer Data Lifecycle QoS policy:

struct WriterDataLifecycleQosPolicy { boolean autodispose\_unregistered\_instances; };

When autodispose\_unregistered\_instances is set to true (the default), a data writer disposes an instance when it is unregistered. In some cases, it may be desirable to prevent an instance from being disposed when an instance is unregistered. This policy could, for example, allow an EXCLUSIVE data writer to gracefully defer to the next data writer without affecting the instance state. Deleting a data writer implicitly unregisters all of its instances prior to deletion.

**Reader\_Data\_Lifecycle:** The READER\_DATA\_LIFECYCLE QoS policy controls the lifecycle of data instances managed by a data reader. Here is the IDL for the Reader Data Lifecycle QoS policy:

struct ReaderDataLifecycleQosPolicy {

Duration\_t autopurge\_nowriter\_samples\_delay;

Duration\_t autopurge\_disposed\_samples\_delay;

};

Normally, a data reader maintains data for all instances until there are no more associated data writers for the instance, the instance has been disposed, or the data has been taken by the user.

The autopurge\_nowriter\_samples\_delay controls how long the data reader waits before reclaiming resources once an instance transitions to the NOT\_ALIVE\_NO\_WRITERS state. By default, autopurge\_nowriter\_samples\_delay is infinite.

The autopurge\_disposed\_samples\_delay controls how long the data reader waits before reclaiming resources once an instance transitions to the NOT\_ALIVE\_DISPOSED state. By default, autopurge\_disposed\_samples\_delay is infinite.

**Time\_Based\_Filter:** The TIME\_BASED\_FILTER QoS policy controls how often a data reader may be interested in changes in values to a data instance. Here is the IDL for the Time Based Filter QoS:

struct TimeBasedFilterQosPolicy { Duration\_t minimum\_separation; };

An interval (minimum\_separation) may be specified on the data reader. This interval defines a minimum delay between instance value changes; this permits the data reader to throttle changes without affecting the state of the associated data writer. By default, minimum\_separation is zero, which indicates that no data is filtered. This QoS policy does not conserve bandwidth as instance value changes are still sent to the subscriber process. It only affects which samples are made available via the data reader.

**Ownership:** The OWNERSHIP policy controls whether more than one Data Writer is able to write samples for the same data-object instance. Ownership can be EXCLUSIVE or SHARED. Below is the IDL related to the Ownership QoS policy:

enum OwnershipQosPolicyKind {

SHARED\_OWNERSHIP\_QOS,

EXCLUSIVE\_OWNERSHIP\_QOS

};

struct OwnershipQosPolicy { OwnershipQosPolicyKind kind; };

If the kind member is set to SHARED\_OWNERSHIP\_QOS, more than one Data Writer is allowed to update the same data-object instance. If the kind member is set to EXCLUSIVE\_OWNERSHIP\_QOS, only one Data Writer is allowed to update a given data-object instance (i.e., the Data Writer is considered to be the owner of the instance) and associated Data Readers will only see samples written by that Data Writer. The owner of the instance is determined by value of the OWNERSHIP\_STRENGTH policy; the data writer with the highest value of strength is considered the owner of the data-object instance. Other factors may also influence ownership, such as whether the data writer with the highest strength is “alive” (as defined by the LIVELINESS policy) and has not violated its offered publication deadline constraints (as defined by the DEADLINE policy).

**Ownership\_Strength:** The OWNERSHIP\_STRENGTH policy is used in conjunction with the OWNERSHIP policy, when the OWNERSHIP kind is set to EXCLUSIVE. Below is the IDL related to the Ownership Strength QoS policy:

struct OwnershipStrengthQosPolicy { long value; };

The value member is used to determine which Data Writer is the owner of the data-object instance. The default value is zero.

**Example**

The following sample code illustrates some policies being set and applied for a publisher.

DDS::DataWriterQos dw\_qos; pub->get\_default\_datawriter\_qos (dw\_qos);

dw\_qos.history.kind = DDS::KEEP\_ALL\_HISTORY\_QOS; dw\_qos.reliability.kind = DDS::RELIABLE\_RELIABILITY\_QOS;

dw\_qos.reliability.max\_blocking\_time.sec = 10;

dw\_qos.reliability.max\_blocking\_time.nanosec = 0;

dw\_qos.resource\_limits.max\_samples\_per\_instance = 100;

DDS::DataWriter\_var dw = pub->create\_datawriter(topic, dw\_qos, 0, // No listener OpenDDS::DCPS::DEFAULT\_STATUS\_MASK);

This code creates a publisher with the following qualities:

• HISTORY set to Keep All

• RELIABILITY set to Reliable with a maximum blocking time of 10 seconds

• The maximum samples per instance resource limit set to 100

This means that when 100 samples are waiting to be delivered, the writer can block up to 10 seconds before returning an error code. These same QoS settings on the Data Reader side would mean that up to 100 unread samples are queued by the framework before any are rejected. Rejected samples are dropped and the SampleRejectedStatus is updated.